【実習】μT-Kernel 入門 (協力：ルネサス エレクトロニクス)

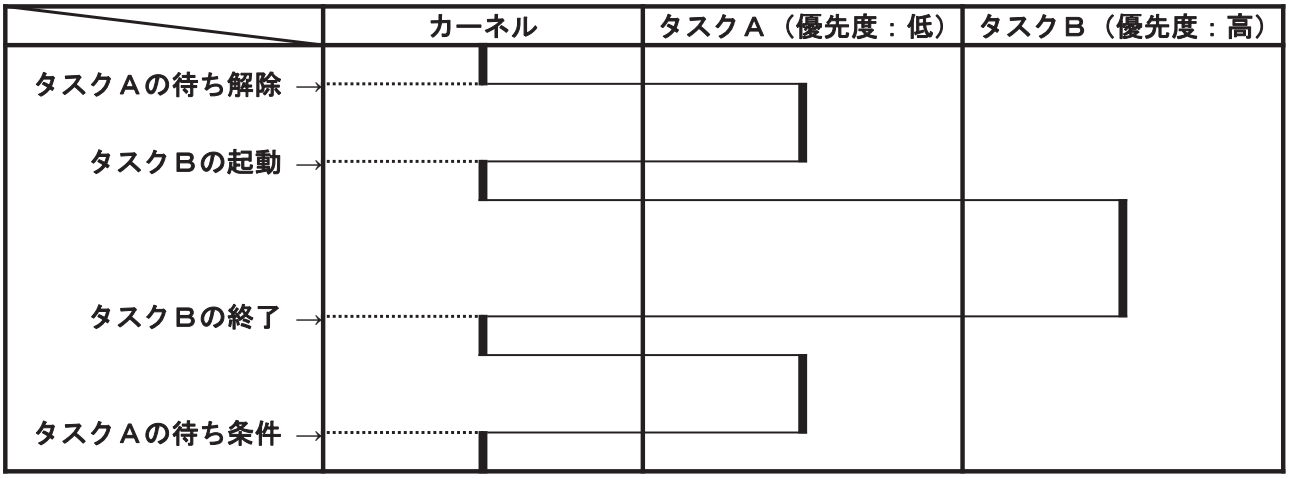
μT-Kernel 入門（補足資料）

TRON Forum

タスクのスケジューリング規則
タスク管理とタスク付属同期

タスクのスケジューリング規則

- T-Kernel仕様におけるタスクのスケジューリング規則はμITRON仕様と同じ。
- 基本は優先度方式、同一優先度はFCFS（First Come First Served）方式。
  優先度の高いタスクが先に実行されるのが特徴である。



タスクのスケジューリングを理解するためには
タスクの状態を先に理解する必要がある

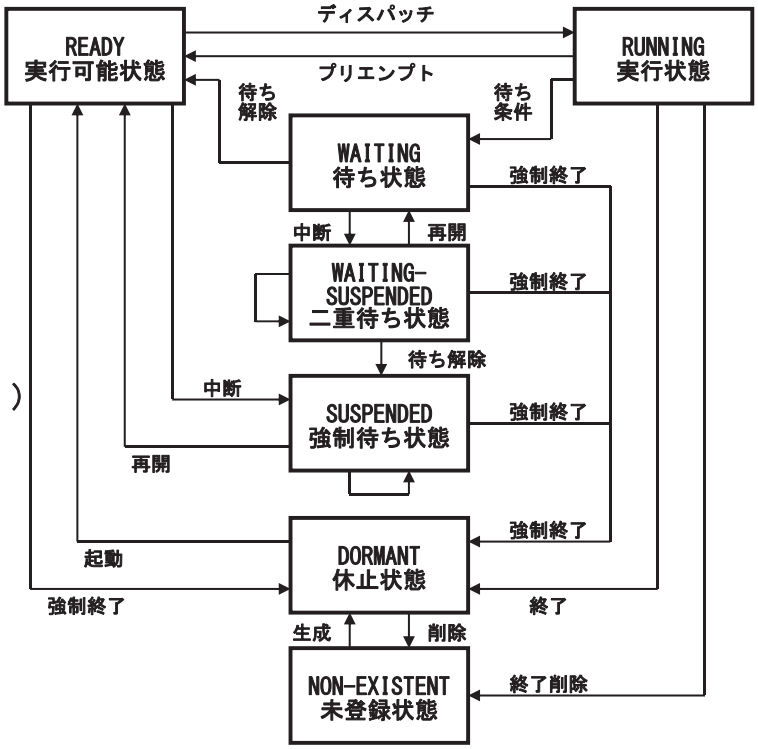
タスクの状態（その1）

■T-Kernel仕様におけるタスクの状態

●T-Kernel仕様におけるタスクの
状態はμITRON4.0仕様と同じ。

- 実行状態（ RUNNING ）
- 実行可能状態（ READY ）
- 広義の待ち状態
  - 待ち状態（ WAITING ）
  - 強制待ち状態（ SUSPENDED ）
  - 二重待ち状態（ WAITING-SUSPENDED ）
- 休止状態（ DORMANT ）
- 未登録状態（ NON-EXISTENT ）

全ての状態を覚える必要はない
先にタスクの基本状態を覚えよう！

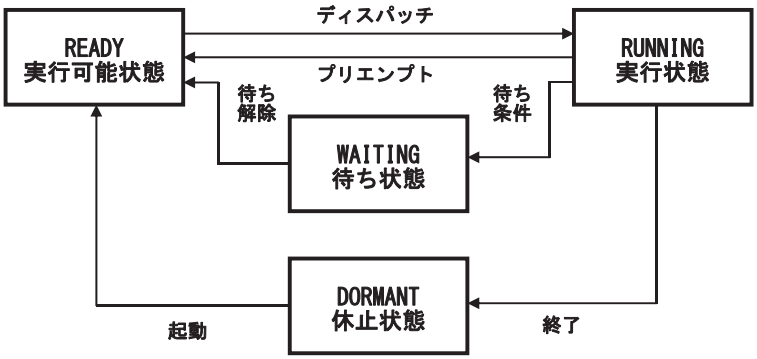


タスクの状態（その2）

■ タスクの基本状態

● 実行状態 （ RUNNING ）
- 与えられた処理を実行している状態

● 実行可能状態 （ READY ）
- 実行する準備は整っているものの、自身のタスクより優先度の高いタスクがRUNNING状態となっているため、実行されないでいる状態

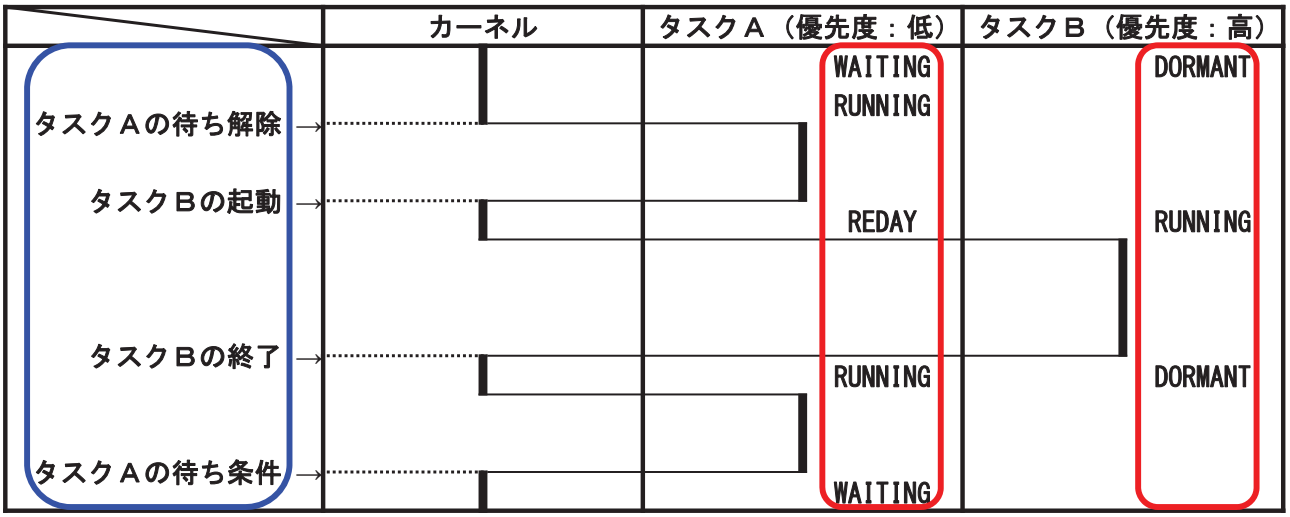


● 待ち状態 （ WAITING ）
- 自身の動作すべき要因が発生するのを待っている状態
- 途中で中断している状態

● 休止状態 （ DORMANT ）
- 自身が起動されるのを待っている状態
- 処理が終了している状態

この４つの状態を覚えれば
殆どのシステムは作成できる！！

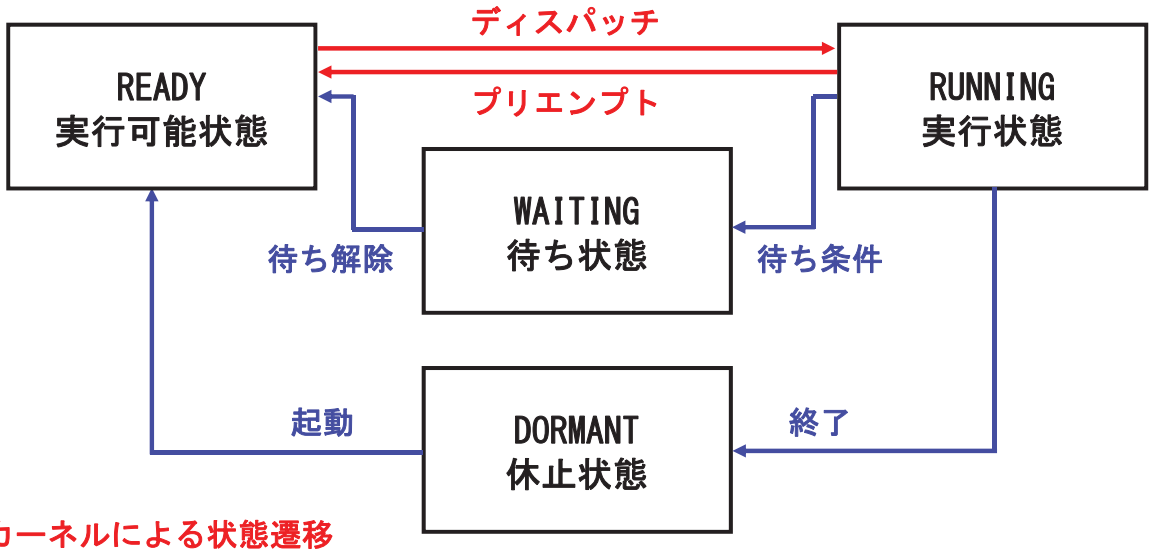
スケジューリングとの対応



タスクの状態を変化させるのが
システムコール（サービスコール）である
T-Kernel仕様　　μITRON4.0仕様

タスクは４つの状態を遷移しながら動作する

状態遷移の要因

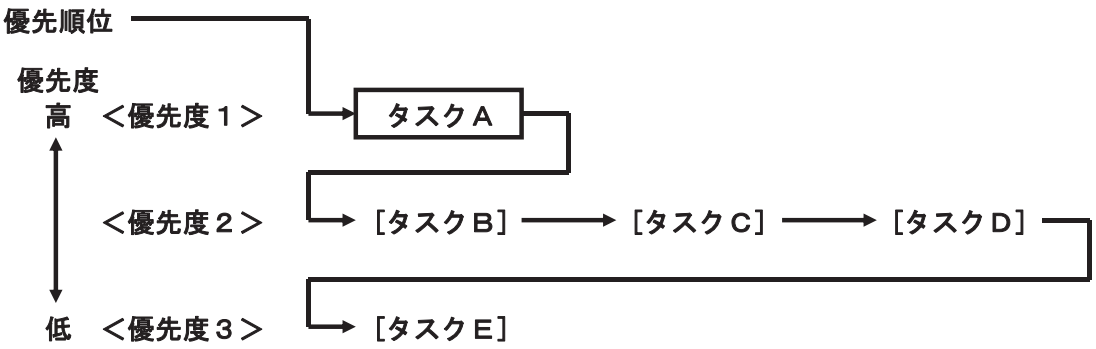


■ カーネルによる状態遷移
- READY状態とRUNNING状態の間の状態遷移
- ディスパッチとプリエンプトはタスクのスケジューリング規則に従いカーネルが実施

■ プログラマの意思による状態遷移
- WAITING状態やDORAMNT状態からREADY状態への遷移
- RUNNING状態からWAITING状態やDORAMNT状態への遷移
- システムコールの発行により、プログラマが制御

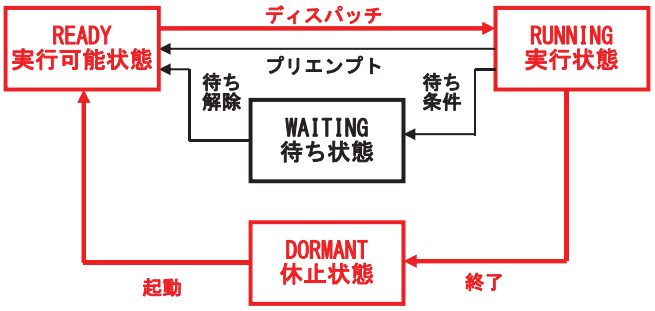
レディキュー

- カーネルがREADY状態の中からRUNNING状態とするタスクを検索するためのキュー。
- WAITING状態やDORMANT状態からREADY状態になるとレディキューに登録される。
  RUNNING状態からWAITING状態やDORMANT状態になるとレディキューから外される。
- 優先度が一番高く、その中で最も先にREADY状態となったタスクがRUNNING状態となる。
  これが優先度方式とFCFS方式の混在型であるタスクのスケジューリング方式。



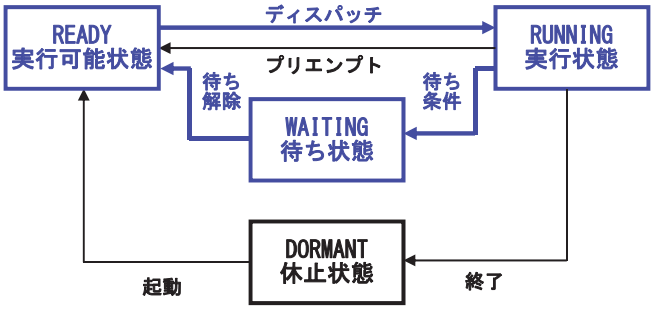
タスクの基本構造

■ タスクは動作する必要がなければレディキューから外れる必要がある。
優先度の高いタスクがRUNNING状態である限り、優先度の低いタスクは動作しない。



● DORMANT状態を利用したタスクの構造

```
void task(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```



● WAITING状態を利用したタスクの構造

```
void task(INT stacd, VP exinf)
{
  while( 1 )  {
    待ちを伴うシステムコール
    // タスクに与えられた処理
    }
}
```


タスクの生成例

```
typedef enum {
  TSK_A, TSK_KIND_NUM
} T_TSK_KIND;
ID gTskid[TSK_KIND_NUM];
          // タスクID番号格納用変数

void create(void)
{
T_CTSK  ctsk = { 0 };
ID      tskid;

  ctsk.exinf   = 0;
  ctsk.tskatr  = TA_HLNG | TA_RNG0;
  ctsk.task    = tsk_a;
  ctsk.itskpri = 10;
  ctsk.stksz   = 1024;

  tskid = tk_cre_tsk( &ctsk );
  if( tskid > E_OK )
    gTskid[TSK_A] = tskid;
}
```

●生成するタスクの記述例

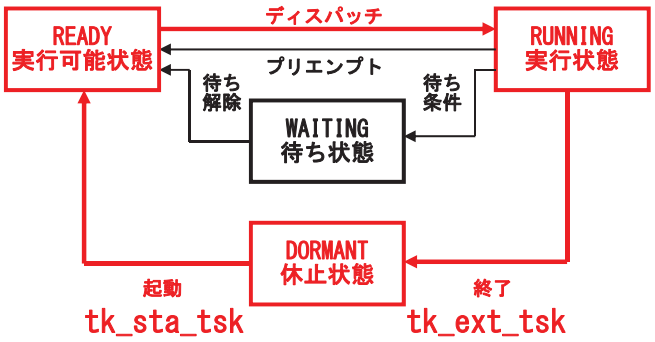
```
void tsk_a(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_ext_tsk( ); // または tk_exd_tsk( );
}
```

```
// タスク生成用のパラメータパケット
// タスクID用の変数

// 拡張情報（タスクの第２引数）の指定は自由
// 高級言語、保護レベル０を指定
// タスクの起動アドレス
// タスクの優先度
// ユーザスタックサイズの指定

// tk_cre_tskシステムコールにより生成
// エラーコードのチェック
// 正常終了なら、ID番号格納用変数に代入
```

タスク管理のシステムコール

タスク生成	ID tskid = tk_cre_tsk(T_CTSK *pk_ctsk);
タスク削除	ER ercd = tk_del_tsk(ID tskid);
タスク起動	ER ercd = tk_sta_tsk(ID tskid, INT stacd);
自タスク終了	void tk_ext_tsk( );
自タスクの終了と削除	void tk_exd_tsk( );
他タスク強制終了	ER ercd = tk_ter_tsk(ID tskid);

■ DORMANT状態を利用したスケジューリングを実現する



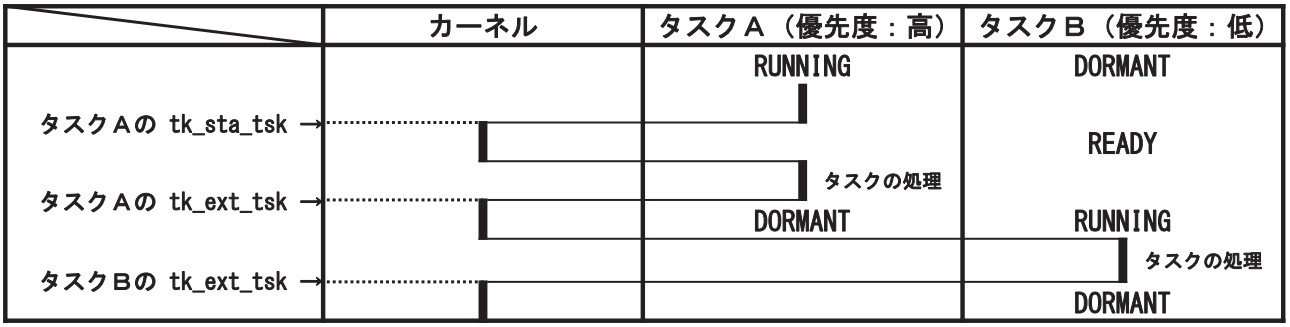

```
void tsk_a(INT stacd, VP exinf)
{
  tk_sta_tsk( gTskid[TSK_B], 0 );
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```

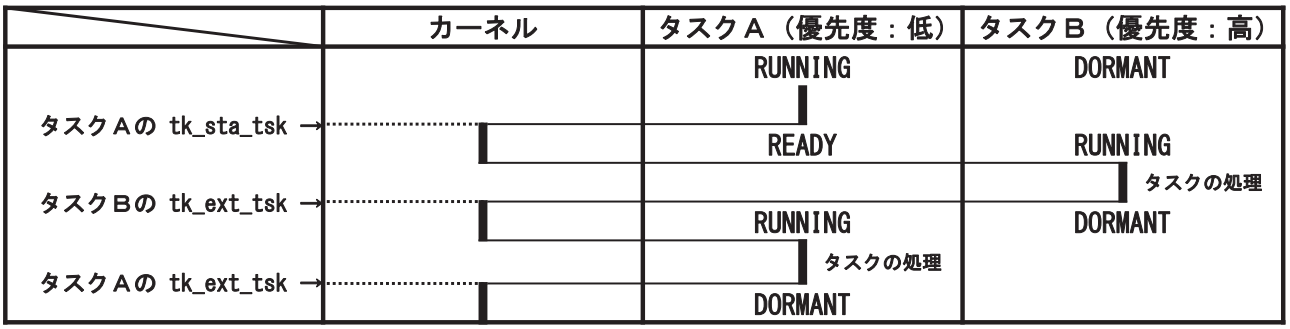
● システムコールのパラメータ
ID tskid → タスクID
INT stacd → 起動コード

タスク管理のスケジューリング

■ 優先度：タスクA＞タスクB



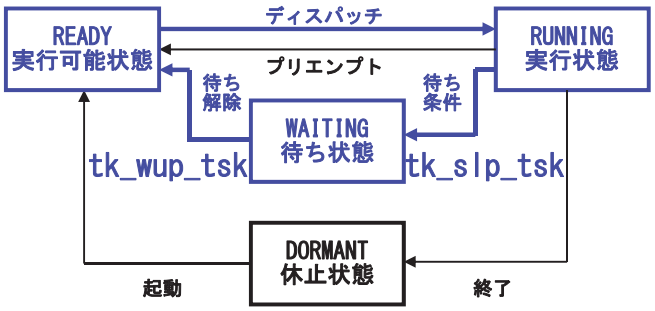
■ 優先度：タスクA＜タスクB



タスク付属同期のシステムコール

自タスクを起床待ち状態へ移行	ER ercd = tk_slp_tsk(TMO tmout);
他タスクの起床	ER ercd = tk_wup_tsk(ID tskid);
タスクの起床要求を無効化	INT wupcnt = tk_can_wup(ID tskid);
他タスクの待ち状態解除	ER ercd = tk_rel_wai(ID tskid);
タスク遅延	ER ercd = tk_dly_tsk(RELTIM dlytim);

■ WAITING状態を利用したスケジューリングを実現する



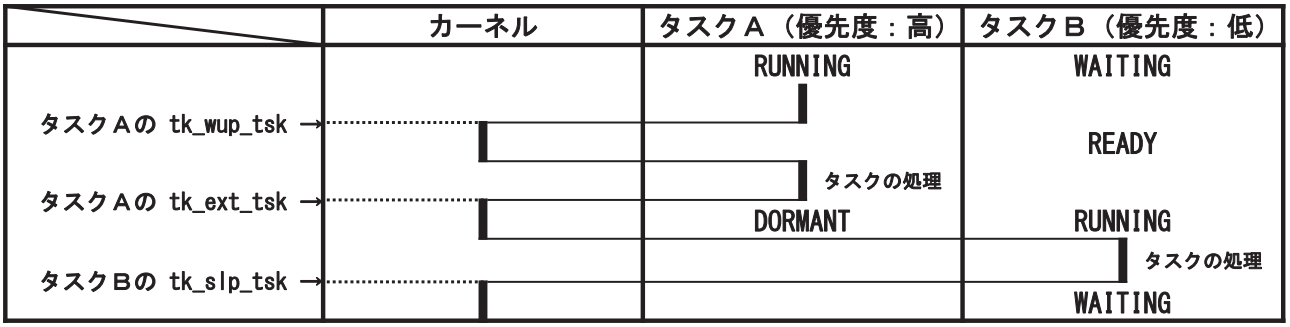

```
void tsk_a(INT stacd, VP exinf)
{
  tk_wup_tsk( gTskid[TSK_B] );
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  while( 1 )  {
    tk_slp_tsk( TMO_FEVR );
    // タスクに与えられた処理
    }
}
```

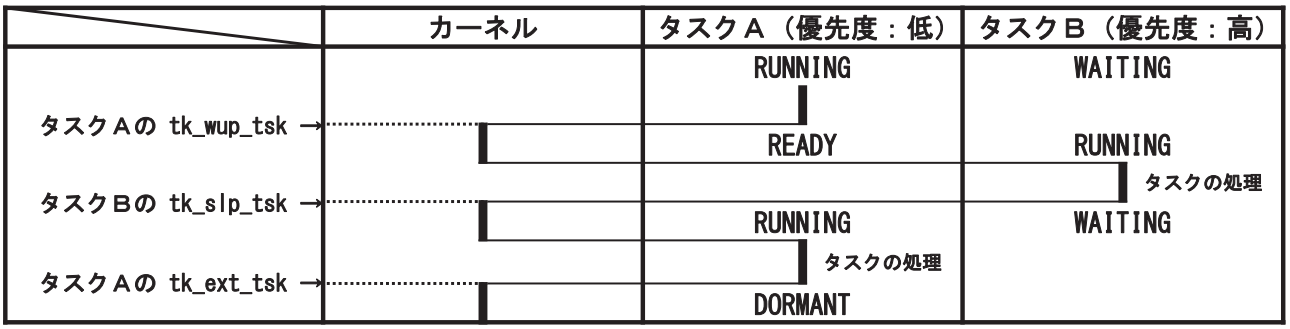
● システムコールのパラメータ
ID　tskid　→　タスクID
TMO　tmout　→　タイムアウト：TMO_FEVRで永久待ち

タスク付属同期のスケジューリング

■ 優先度：タスクA＞＝タスクB



■ 優先度：タスクA＜タスクB



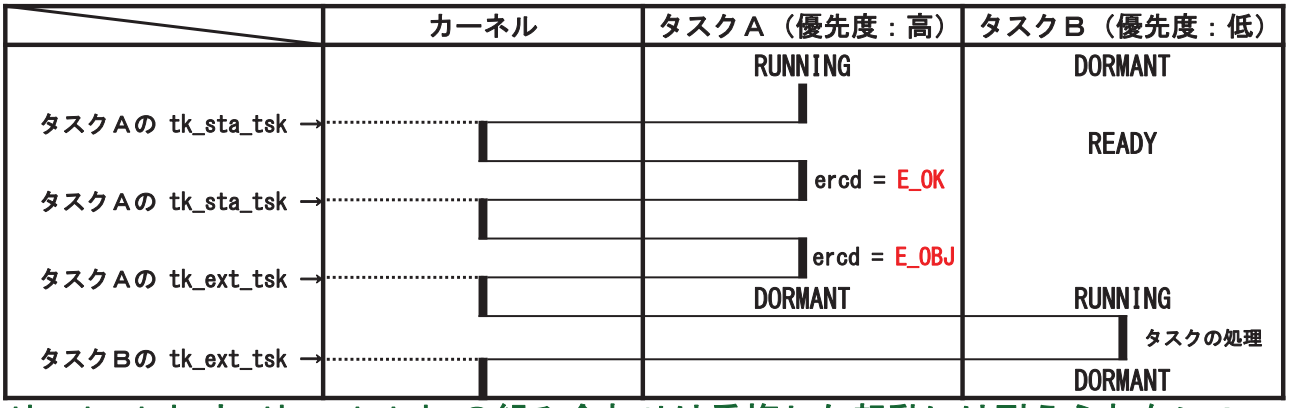
タスク管理とタスク付属同期の違い

```
void tsk_a(INT stacd, VP exinf)
{
ER ercd;
  ercd = tk_sta_tsk( gTskid[TSK_B], 0 );
  ercd = tk_sta_tsk( gTskid[TSK_B], 0 );
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```

● システムコールのエラーコード
E_OK → 正常終了
E_OBJ → オブジェクト状態不正

■ 優先度：タスクＡ＞＝タスクＢ



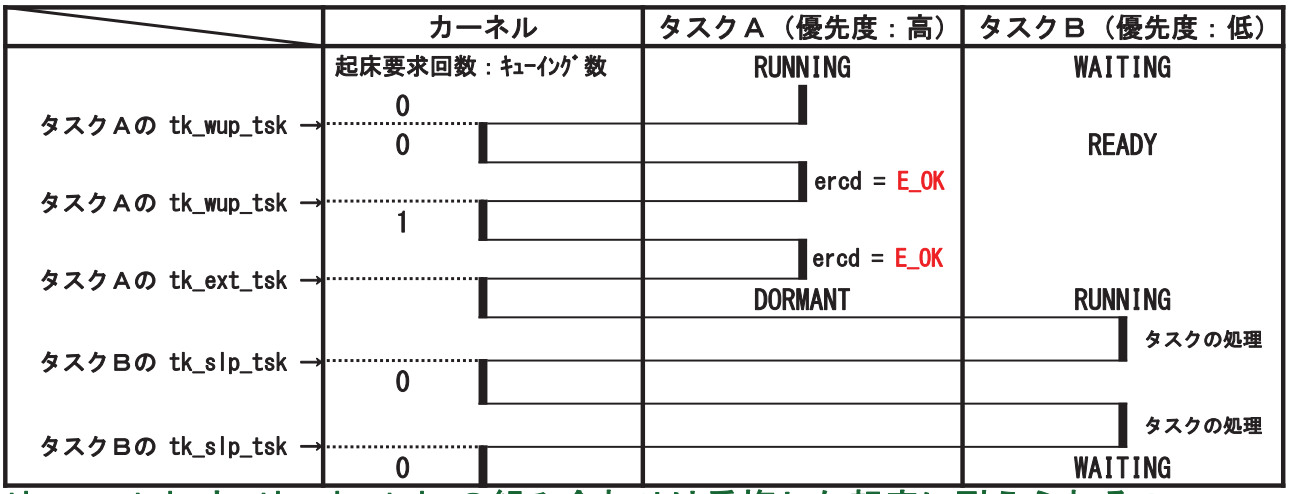
tk_sta_tsk と tk_ext_tsk の組み合わせは重複した起動には耐えられない！

タスク管理とタスク付属同期の違い

```
void tsk_a(INT stacd, VP exinf)
{
ER ercd;
  ercd = tk_wup_tsk( gTskid[TSK_B] );
  ercd = tk_wup_tsk( gTskid[TSK_B] );
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  while( 1 )  {
    tk_slp_tsk( TMO_FEVR );
    // タスクに与えられた処理
    }
}
```

■ 優先度：タスクＡ＞＝タスクＢ



tk_wup_tsk と tk_slp_tsk の組み合わせは重複した起床に耐えられる！

タスク管理とタスク付属同期の違い

■ tk_sta_tsk は起動コードが渡せる（実行＋情報）。
起動コードを利用し、同一のプログラムを複数のタスクとして生成した例。

```
void create(void)
{
T_CTSK  ctsk = { 0 };                  // タスク生成用のパラメータパケット
ID      tskid;          int i;         // タスクID用の変数
  ctsk.tskatr  = TA_HLNG | TA_RNG0;    // 高級言語、保護レベル０を指定
  ctsk.task    = (FP) tsk;             // タスクの起動アドレス
  ctsk.itskpri = 10;                   // タスクの優先度
  ctsk.stksz   = 1024;                 // ユーザスタックサイズの指定
  for( i=0 ; i<2 ; i++ )  {
    tskid = tk_cre_tsk( &ctsk );       // tk_cre_tskシステムコールにより生成
    if( tskid > E_OK )                 // エラーコードのチェック
      tk_sta_tsk( tskid, (INT)i );     // 起動コードで各タスクの違いを知らせる
    }
}
```

```
void tsk(INT stacd, VP exinf)
{       //    0
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```

```
void tsk(INT stacd, VP exinf)
{       //    1
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```


タスク管理とタスク付属同期の違い

■ DORMANT状態はタスクが終了した状態（局所変数は削除される）。
WAITING状態はタスクが中断した状態（局所変数の値は保持されている）。

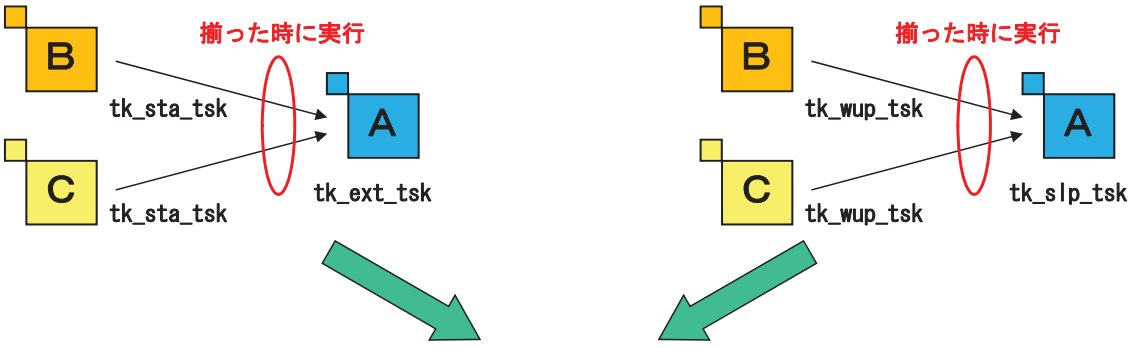
```
void tsk(INT stacd, VP exinf)
{
int work = 0;              // 局所変数を初期化

  work = 100;              // 設定値は tk_ext_tsk により失われる
  tk_ext_tsk( );           // tk_sta_tsk による再起動時は初期値に戻る
}
```

```
void tsk(INT stacd, VP exinf)
{
int work = 0;              // 局所変数を初期化

  while( 1 )  {

    work = 100;            // tk_slp_tsk を発行しても設定値は失われない
    tk_slp_tsk( );         // tk_wup_tsk による起床時は 100 のまま
    }
}
```

イベントフラグ

イベントフラグの概要

■ イベントフラグは複数のイベントが揃った時に特定のタスクを実行するものである。
例えば、タスク管理やタスク付属同期の機能では上記の機能は実現できない。



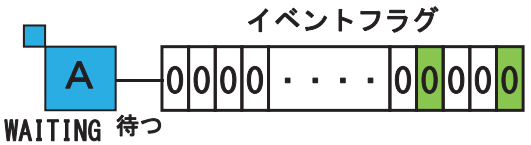
どちらも不可能

誰かが tk_sta_tsk を発行すれば、tk_ext_tsk を発行済みのタスクは動作する
誰かが tk_wup_tsk を発行すれば、tk_slp_tsk を発行済みのタスクは動作する

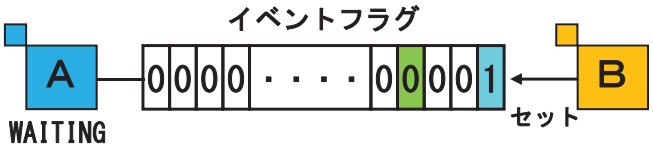
これを実現するのがイベントフラグであると考えれば良い

イベントフラグの概要

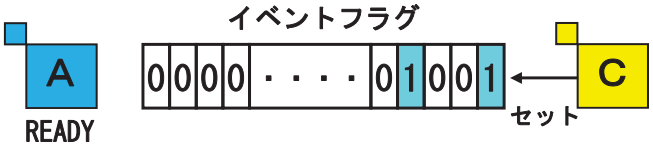
■ イベントフラグはイベントの発生を１ビットのフラグで管理することにより、タスク間で同期処理を行うものである。（T-Kernel仕様ではイベントフラグは32ビット以上）



イベントフラグで待つタスクは待つビットを指定する。
カーネルは指定されたビットがセットされるまで、目的のタスクをWAITING状態とする。



イベントを通知するタスクはセットするビットを指定する。
ただし、待っているビットが全てセットされなければ、
目的のタスクのWAITING状態は解除されない。 （AND待ち）



待っているビットが全てセットされなければ、
目的のタスクのWAITING状態は解除され、READY状態となる。
READY状態となればスケジューリング規則に従って動作する。

イベントフラグのシステムコール

```
イベントフラグ生成       ID flgid = tk_cre_flg( T_CFLG *pk_cflg );
イベントフラグ削除       ER ercd  = tk_del_flg( ID flgid );
イベントフラグのセット   ER ercd  = tk_set_flg( ID flgid, UINT setptn );
イベントフラグのクリア   ER ercd  = tk_clr_flg( ID flgid, UINT clrptn );
イベントフラグ待ち       ER ercd  = tk_wai_flg( ID flgid, UINT waiptn,
                                        UINT wfmode, UINT *p_flgptn, TMO tmout );
イベントフラグ状態参照   ER ercd  = tk_ref_flg( ID flgid, T_RSEM *pk_rflg );
```

● システムコールのパラメータ

```
ID    flgid   →  イベントフラグID
UINT  setptn  →  セットするビットパターン（"1" を指定したビットをセットする）
UINT  waiptn  →  待つビットパターン　　（"1" を指定したビットを待つ）
UINT  wfmode  →  待ちモード：（TWF_ANDW // TWF_ORW）| [ TWF_CLR // TWF_BITCLR ]
TMO   tmout   →  タイムアウト：TMO_FEVRで永久待ち
```

● 待ちモード（AND待ちとOR待ち）
- TWF_ANDW ：AND待ち → waiptnで指定した全てのビットがセットされるのを待つ
- TWF_ORW ：OR待ち → waiptnで指定したビットのいづれかがセットされるのを待つ

● クリア指定
- 未指定 ：待ち解除時のビットクリアは行わない
- TWF_CLR ：待ち解除時に全てのビットをクリア
- TWF_BITCLR：待ち解除時にwaiptnで指定したビットのみクリア

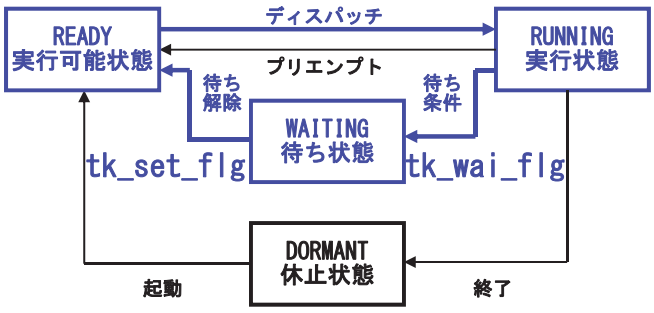
イベントフラグのコーディング例

```
void tsk_a(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_set_flg( gFlgid[FLG_X], 0x0001 );
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_set_flg( gFlgid[FLG_X], 0x0002 );
  tk_ext_tsk( );
}
```

■ タスクの優先度に依存せず、必ずタスクCの処理を最後に行うコーディング

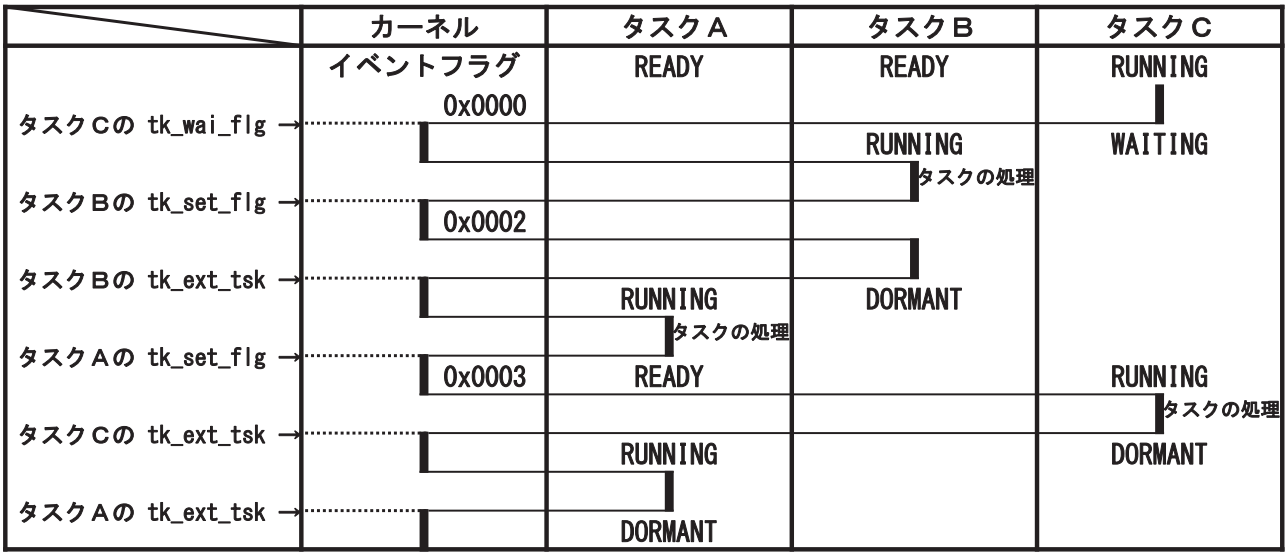
■ WAITING状態を利用したスケジューリングを実現する




```
void tsk_c(INT stacd, VP exinf)
{
UINT flgptn;
  tk_wai_flg( gFlgid[FLG_X], 0x0003, TWF_ANDW, &flgptn, TMO_FEVR );
  // タスクに与えられた処理
  tk_ext_tsk( );
}
```

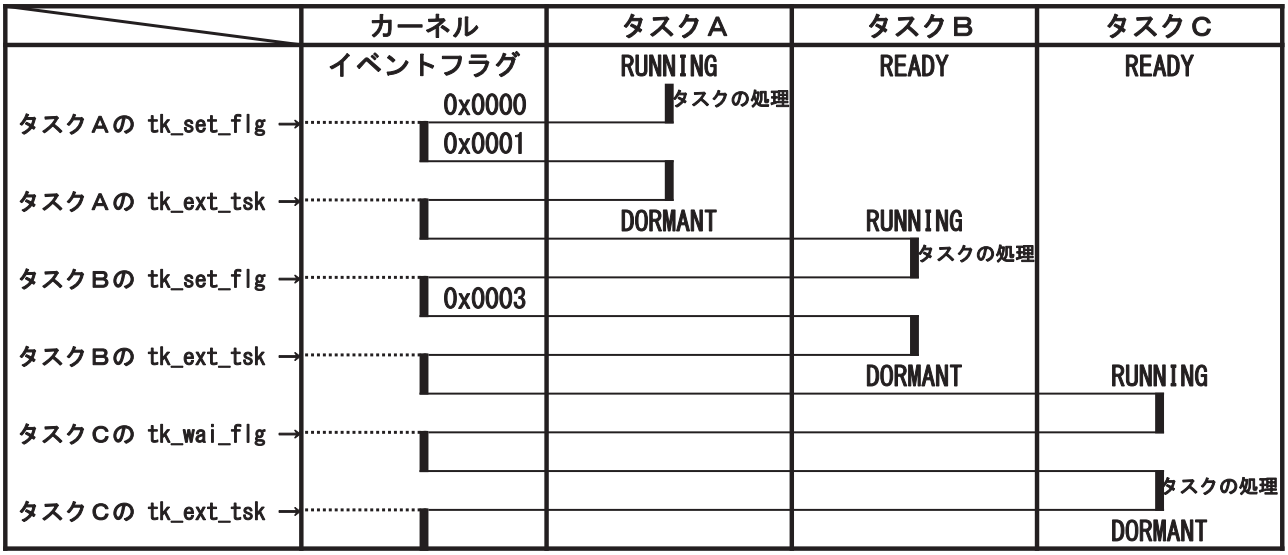
イベントフラグのスケジューリング（その1）

■ 優先度：タスクＡ＜タスクＢ＜タスクＣ



イベントフラグのスケジューリング（その2）

■ 優先度：タスクＡ＞タスクＢ＞タスクＣ



イベントフラグのクリア方法

■ イベントフラグにセットされたビットはクリアする意思を持たなければクリアされない。
クリアする方法は２つ用意されている。

- tk_clr_flgシステムコールを使ったクリア方法
- tk_wai_flgシステムコール発行時の待ちモード（wfmode）を使ったクリア方法

■ tk_clr_flgシステムコールは clrptn で指定した値と現在のイベントフラグの値で
AND（論理積）を取ってクリアを行う。

イベントフラグのクリア　ER ercd　= tk_clr_flg(ID flgid, UINT clrptn);

例：　現在のイベントフラグ　AND　clrptn　＝　クリア後のイベントフラグ
　　0x1234　&　0x00FF　＝　0x0034

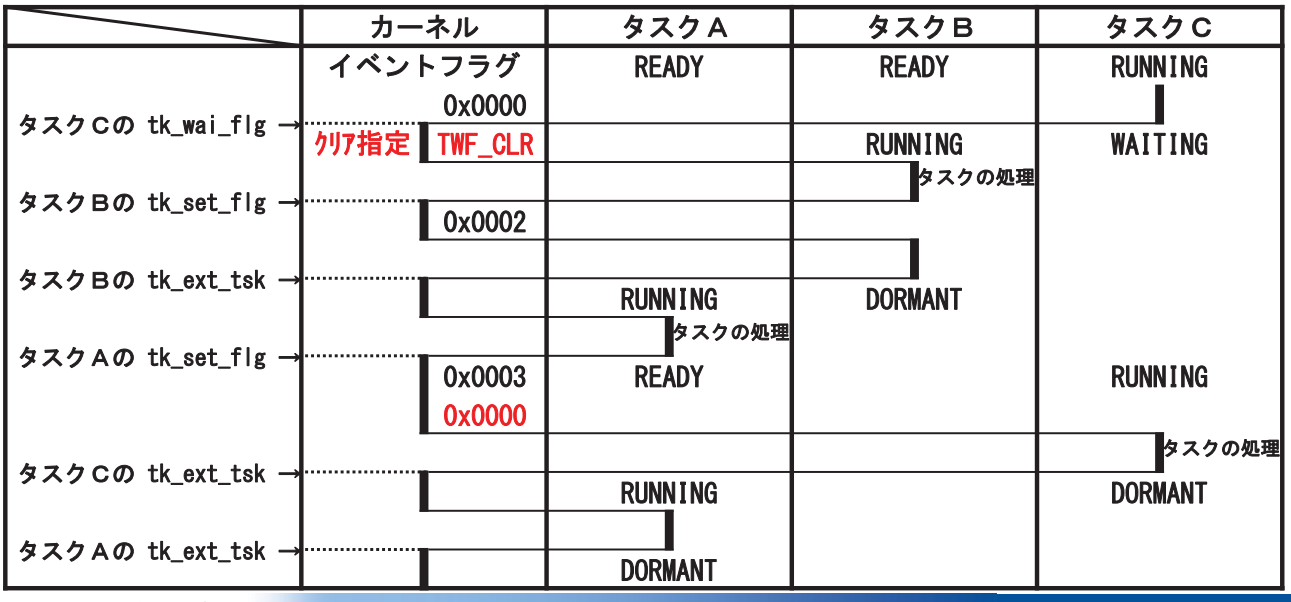
```
void tsk_c(INT stacd, VP exinf)
{
UINT flgptn;
  while( 1 )  {
    tk_wai_flg( gFlgid[FLG_X], 0x0003, TWF_ANDW, &flgptn, TMO_FEVR );
    // タスクに与えられた処理
    tk_clr_flg( gFlgid[FLG_X], ~0x0003 );  // waiptnに ~ を付けてclrptnを指定
    }                                      // 意味：自身の待ち要因のみクリアする
}
```


イベントフラグのクリア方法

■ tk_wai_flgシステムコール発行時の待ちモード（wfmode）として、TWF_CLR か TWF_BITCLR
の指定を行えば、カーネルによって目的のタスクが待ち解除時に自動的にクリアを行う。

クリア指定　－ 未指定　　：待ち解除時のビットクリアは行わない
　　－ TWF_CLR　：待ち解除時に全てのビットをクリア
　　－ TWF_BITCLR：待ち解除時にwaiptnで指定したビットのみクリア

■ 優先度：タスクＡ＜タスクＢ＜タスクＣ



tk_wai_flgのリターンパラメータ

■ tk_wai_flgシステムコールの第４引数はリターンパラメータである。
リターンパラメータには待ち解除時のイベントフラグの内容が渡される。
なお、クリア指定がある場合はクリア直前のイベントフラグの内容が渡される。

```
イベントフラグ待ち      ER ercd  = tk_wai_flg( ID flgid, UINT waiptn,
                                    UINT wfmode, UINT *p_flgptn, TMO tmout );
```

■ 待ちモードがAND待ち（TWF_ANDW）の時は意味を成さない。

```
void tsk_a(INT stacd, VP exinf)
{
  tk_set_flg( gFlgid[FLG_X], 0x0001 );
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  tk_set_flg( gFlgid[FLG_X], 0x0002 );
  tk_ext_tsk( );
}
```

```
void tsk_c(INT stacd, VP exinf)
{
UINT flgptn;
  tk_wai_flg( gFlgid[FLG_X], 0x0003, TWF_ANDW, &flgptn, TMO_FEVR );
  tk_ext_tsk( );
}
```

waiptnと同じはず

tk_wai_flgのリターンパラメータ

■ 待ちモードがOR待ち（TWF_ORW）ならば、待ち解除要因の確認に使用可能。

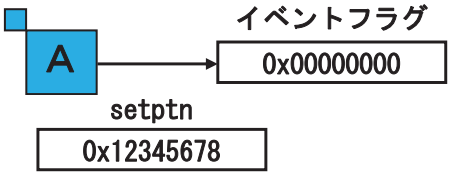
```
void tsk_a(INT stacd, VP exinf)
{
  tk_set_flg( gFlgid[FLG_X], 0x0001 );
  tk_ext_tsk( );
}
```

```
void tsk_b(INT stacd, VP exinf)
{
  tk_set_flg( gFlgid[FLG_X], 0x0002 );
  tk_ext_tsk( );
}
```

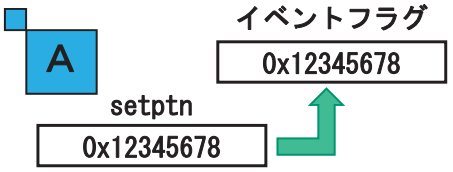
```
void tsk_c(INT stacd, VP exinf)
{
UINT flgptn;
  tk_wai_flg( gFlgid[FLG_X], 0x0003, TWF_ORW, &flgptn, TMO_FEVR );

  if( (flgptn & 0x0001) != 0 )
    // タスクＡからのtk_set_flgに対する処理を実行
  if( (flgptn & 0x0002) != 0 )
    // タスクＢからのtk_set_flgに対する処理を実行

  tk_ext_tsk( );
}
```

イベントフラグによるデータの受け渡し

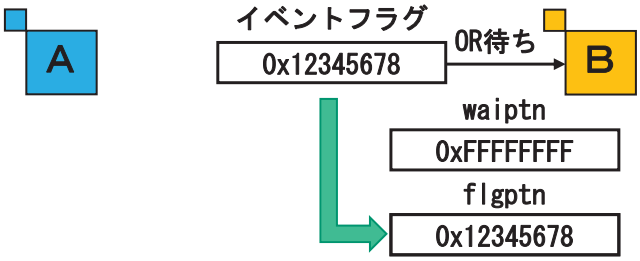
■ イベントフラグはリターンパラメータを使って、タスク間、タスクと割込みハンドラの間でデータの受け渡しを行うことが可能である。



タスク（割込みハンドラ）は渡したいデータをsetptnとして、tk_set_flgを発行する。



setptnに設定した渡したいデータはセットするビットとして、イベントフラグに設定される。



データを受け取りたいタスクはwaiptnで全ビットを指定し、OR待ちでtk_wai_flgを発行する。そうすると渡したいデータがゼロ以外であれば、イベントフラグの内容（渡したいデータ）がリターンパラメータ（flgptn）に返される。

イベントフラグによるデータの受け渡し

■ イベントフラグを使った割込みハンドラからタスクへのデータの受け渡し例。

```
void SCI_hdr(void)
{
  tk_set_flg( gFlgid[FLG_X], SCI.RDR );     // SCI.RDRは受信データレジスタ
}                                           // 受信データをイベントフラグに設定
```

```
void rcv_tsk(INT stacd, VP exinf)
{
UINT flgptn;
  while( 1 )  {
    tk_wai_flg( gFlgid[FLG_X], 0xFF, TWF_ORW, &flgptn, TMO_FEVR );
    // flgptnに返された受信データ（１バイト）を使って処理を実行
    }
}
```

イベントフラグの生成

```
typedef enum {
  FLG_X, FLG_KIND_NUM
} T_FLG_KIND;

static ID gFlgid[FLG_KIND_NUM];          // イベントフラグID番号格納用変数

void create(void)
{
T_CFLG  cflg = { 0 };                    // イベントフラグ生成用のパラメータパケット
ID      flgid;                           // イベントフラグID用の変数

  cflg.exinf = 0;                        // 拡張情報の指定は自由
  cflg.flgatr = (TA_TFIFO | TA_WSGL);    // イベントフラグの属性を指定
  cflg.iflgptn = 0;                      // イベントフラグの初期ビットパターンを指定

  flgid = tk_cre_flg( &cflg );           // tk_cre_flgシステムコールにより生成
  if( flgid > E_OK )                     // エラーコードのチェック
    gFlgid[FLG_X] = flgid;               // 正常終了なら、ID番号格納用変数に代入
}
```

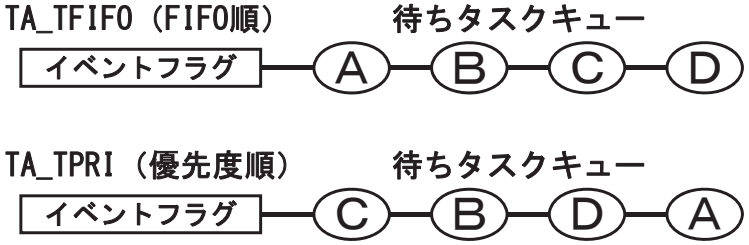

イベントフラグの属性

■ イベントフラグを生成時は属性として、以下の２つの項目を指定する。
- 「待ちタスクキューの並び方」　→　マクロ名：TA_TFIFO か TA_TPRI を指定
- 「複数待ちタスクの許可」　→　マクロ名：TA_WSGL か TA_WMUL を指定

■ 「複数待ちタスクの許可」は、１つのイベントフラグに複数のタスクが tk_wai_flg を使って待つことを許すかどうかを指定する。
- TA_WSGL　→　許さない。待てるタスクは１つのみ。
- TA_WMUL　→　許す。複数のタスクが待てる。

■ 「待ちタスクキューの並び方」は、複数待ちタスクを許可した場合の待ちタスクキューの並び方を指定する。
- TA_TFIFO　→　FIFO順：tk_wai_flg を発行した順番に待ちタスクキューに登録する。
- TA_TPRI　→　優先度順：タスクの優先度順に待ちタスクキューに登録する。

tk_wai_flgの発行順	タスクの優先度
タスクA	低
タスクB	中
タスクC	高
タスクD	中



「待ちタスクキューの並び方」は他のカーネル・オブジェクトにもあるため、覚えておくと良い

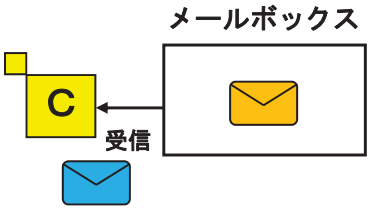
メールボックスとメッセージバッファ

メールボックスの概要

■ メールボックスはタスク間でデータの受け渡しを行いながら同期処理を実現します。
送受信するメッセージはコピーせず、そのアドレスのみを受け渡すのが特徴です。

送信を行うタスクは、メッセージ形式でデータを作成し、それをメールボックスに送信する。送信されたメッセージはメールボックス内に入る。
なお、メッセージを送信したタスクがWAITING状態になることはない。

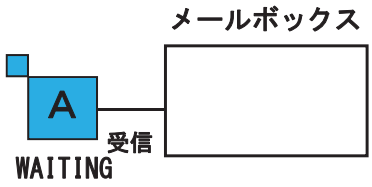
既にメッセージが送信されているメールボックスに対し、別のメッセージを送信すれば、メールボックス内でメッセージは待ち行列で管理される。



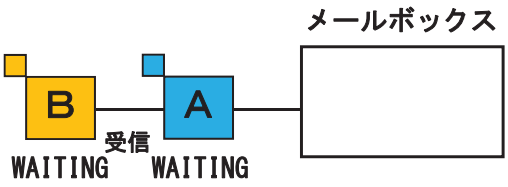
メールボックス内にメッセージがある状態で受信を行えば、待ち行列先頭のメッセージが受信を行ったタスクに渡される。
なお、メールボックス内にメッセージがある限り、受信を行ったタスクがWAITING状態になることはない。

メールボックスの概要

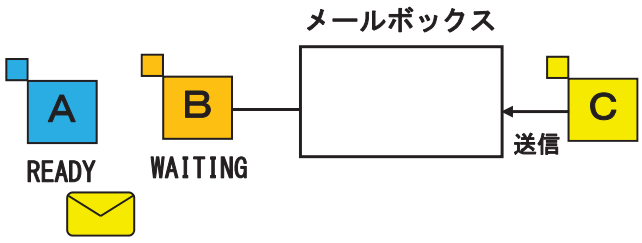
■ メールボックスはタスク間でデータの受け渡しを行いながら同期処理を実現します。
送受信するメッセージはコピーせず、そのアドレスのみを受け渡すのが特徴です。



メッセージが送信されていないメールボックスから、メッセージの受信を行うと、そのタスクはメッセージ受信待ちのWAITING状態となる。



既にメッセージ受信待ちのタスクが存在するメールボックスから、メッセージの受信を行うと、そのタスクもメッセージ受信待ちのWAITING状態となり、タスクは待ち行列で管理される。

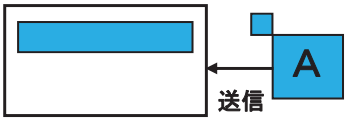


メッセージ受信待ちのタスクは、メッセージが送信されれば、そのメッセージを受信し、メッセージ受信待ちのWAITING状態からREADY状態となる。

メッセージバッファの概要

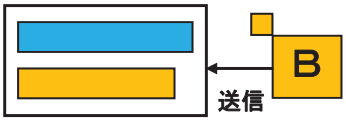
■ メッセージバッファもタスク間でデータの受け渡しを行いながら同期処理を実現します。
メールボックスとは異なり、送受信するメッセージはコピーされるのが特徴です。

メッセージバッファ



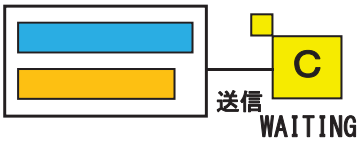
メッセージバッファに対して送信を行うと、メッセージはコピーされてメッセージバッファに入る。

メッセージバッファ



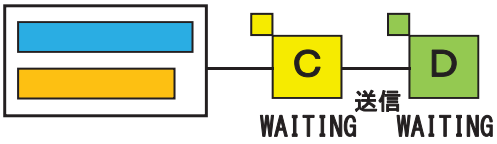
メッセージバッファに空きがある限り、送信を行ったタスクがWAITING状態になることはない。

メッセージバッファ



メッセージバッファに空きがない状態の時に送信を行ったタスクは、送信待ちのWAITING状態となる。

メッセージバッファ

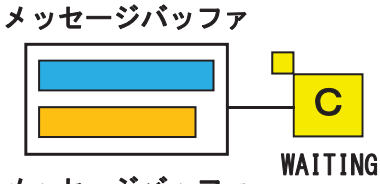


既に送信待ちのタスクが存在する時に送信を行うと、そのタスクも送信待ちのWAITING状態となり、送信待ちのタスクは待ち行列で管理される。

メッセージバッファの概要

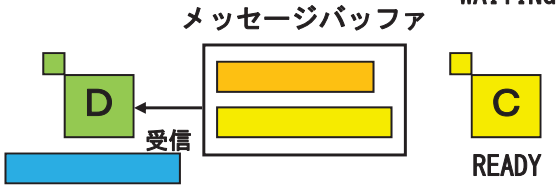
■ メッセージバッファもタスク間でデータの受け渡しを行いながら同期処理を実現します。
メールボックスとは異なり、送受信するメッセージはコピーされるのが特徴です。

メッセージバッファ



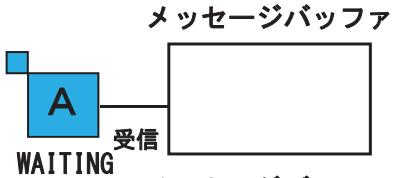
メッセージバッファに空きがない状態の時に送信を行ったタスクは、送信待ちのWAITING状態となる。

メッセージバッファ



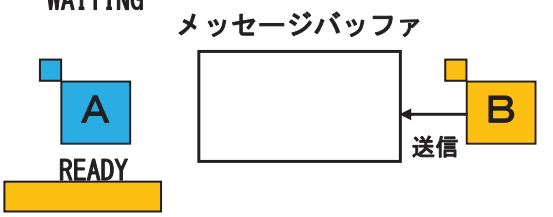
送信待ちのWAITITNG状態となったタスクは、メッセージが受信されてメッセージバッファに空きができ、送信データがコピーできるようになれば、メッセージ送信待ちのWAITING状態からREADY状態となる。

メッセージバッファ



メッセージが送信されていないメッセージバッファから、メッセージの受信を行ったタスクは、メッセージ受信待ちのWAITING状態となる。

メッセージバッファ

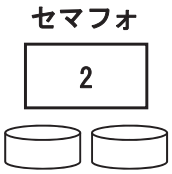


メッセージ受信待ちのWAITING状態となったタスクは、メッセージが送信されれば、それを受信し、メッセージ受信待ちのWAITING状態からREADY状態となる。

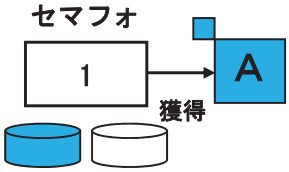
セマフォとミューティクス

セマフォの概要

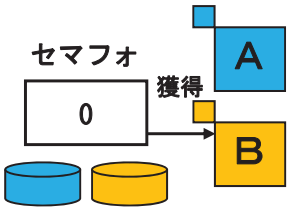
■ セマフォは資源数を示すセマフォカウンタで共有資源の排他制御を実現します。



セマフォカウンタは、獲得可能な資源数を示す。
資源の獲得・返却のシステムコールに合わせて増減が行われる。



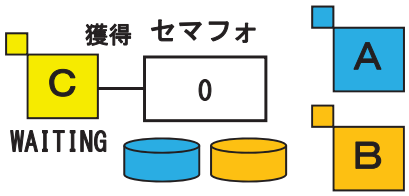
資源の獲得を要求すると、空いている資源があれば（セマフォカウンタ≧要求数）、資源を獲得し、獲得した資源の個数分、セマフォカウンタは減少する。
なお、実際に獲得できた資源がどれであるかは、セマフォとは別の何かで管理する必要がある。



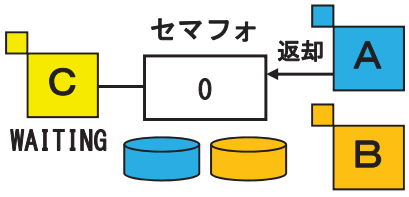
空いている資源があれば（セマフォカウンタ≧要求数）、動作は同じ。
また、資源が獲得できる限り、タスクがWAITING状態となることはない。

セマフォの概要

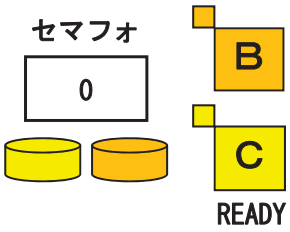
■ セマフォは資源数を示すセマフォカウンタで共有資源の排他制御を実現します。



空いている資源がなければ（セマフォカウンタ＜要求数）、資源の獲得を要求したタスクは、資源獲得待ちのWAITING状態となる。
この状態で更に別のタスクが資源の獲得を要求し、資源の獲得待ちとなれば、そのタスクも資源獲得待ちのWAITING状態となる、待ち行列で管理される。



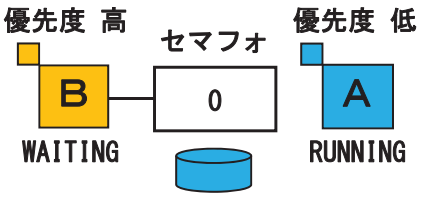
資源獲得待ちのWAITING状態となっているタスクは、資源が返却され、資源が獲得可能（セマフォカウンタ≧要求数）となれば、資源を獲得してREADY状態となる。



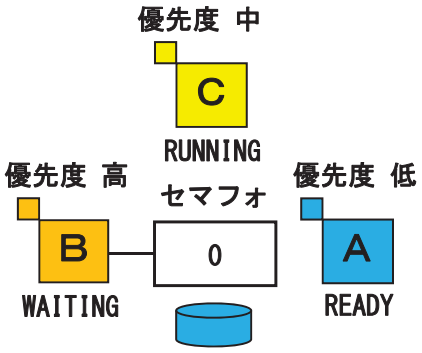
なお、資源を返却した際、資源獲得待ちのタスクが存在しなければ、返却した資源数分、セマフォカウンタは増加する。

セマフォの優先度逆転現象

■ 資源獲得待ちのタスクの優先度は、資源獲得中のタスクの優先度に左右されてしまう優先度逆転現象という欠点がセマフォにはある。



優先度が高くても、WAITING状態となったタスクの優先度は考慮されない。
このため、資源を獲得している優先度の低いタスクが資源を返却するまで、資源獲得待ちであるWAITING状態のタスクは実行されない。

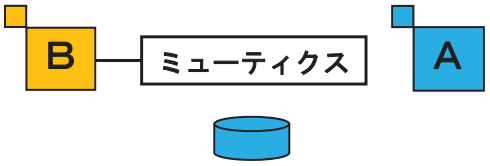


この時、資源を獲得しているタスクより優先度は高く、セマフォとは無関係のタスクがREADY状態になると、そのタスクがRUNNING状態となり、資源を獲得しているタスクはプリエンプトされてREADY状態となる。
この中で一番優先度が高いのは資源獲得待ちのタスクであるが、資源を獲得しているタスクの優先度に引きずられ、実行されない状態になっている。

ミューティクスの概要

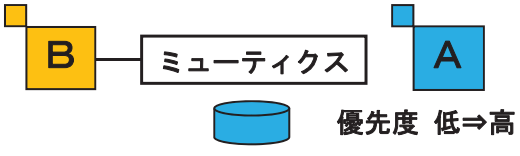
■ ミューティクスもセマフォと同じく資源の排他制御に利用します。
ミューティクスには３つの使い方があり、生成時の属性によって使い方が決定します。

● 属性がTA_TFIFO（FIFO順）またはTA_TPRI（優先度順）



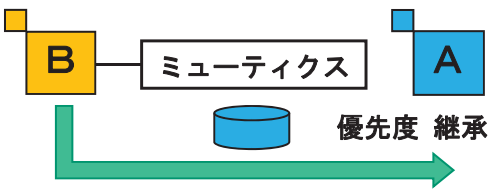
資源数が 1 のセマフォ（バイナリセマフォ）と同じ動作です。
バイナリセマフォの場合、優先度逆転現象は回避できません。

● 属性がTA_CEILING（優先度上限プロトコル）



ミューティクスをロック（資源を占有）した時点で目的のタスクの優先度を上限まで（生成時で指定した値まで）引き上げます。
資源をロックすると優先度が変化するため、優先度逆転現象が回避できます。

● 属性がTA_INHERIT（優先度継承プロトコル）



ミューティクスをロック（資源を占有）中、自身より優先度の高いタスクが資源のロック待ちとなった際、そのタスクの優先度を継承します。
資源ロック待ちのタスクの優先度を継承するため、これも優先度逆転現象を回避できます。

トロンフォーラム
【実習】μT-Kernel 入門(協力:ルネサス エレクトロニクス)
μT-Kernel 入門（補足資料）

2016 年 6 月 23 日発行

発行所
トロンフォーラム
(YRP ユビキタス・ネットワーキング研究所内)
〒141-0031 東京都品川区西五反田 2-12-3 第一誠実ビル 9F
URL: http://www.tron.org/ja/
TEL:03-5437-0572(代表) FAX:03-5437-2399(代表)

トロンフォーラム©2016
Printed in Japan